Composition via Sonification of Wave Buoy Data

Ethan Bailey - ehb282@nyu.edu

Abstract

This program creates an evolving piece of music using data obtained from a specific NOAA Data Buoy in the middle of the Cape Cod Bay. The three performing instruments are controlled entirely by various data that the buoy captures in real time, and an additional delay effect is also controlled by similar data. The piece serves as a sonic representation of about a month and a half of weather conditions in the center of the bay, and it continues without repeating for a little over two hours.

Intro & Detailed Description

Having grown up on Cape Cod in Massachusetts, I've always felt attached to the ocean. The oceans surrounding the Cape caused the weather to be generally congruous with the conditions of the seas - if it was stormy on the water, it was stormy on land as well. I've tried to incorporate imagery from this rather turbulent natural landscape in my more traditional compositions, so for this algorithmic composition, I decided I'd take the real data representing the condition of the seas and turn it into music. The NOAA National Data Buoy Center publishes the data collected by every buoy in their system online for free, and each buoy's information is collected in intervals of 30 minutes. In my piece, each 30 minute segment lasts about 15 seconds.

There are three primary instruments in the piece, and one effects generator: two "chord" synths, a bass synth, and a delay effect. The chord synths are operated independently, and are driven by two different types of waves that the buoy measures. I didn't want the data to directly determine each note in the piece, as that would result in a rather non-musical distribution of notes - though I wanted to be able to employ dissonance, I also wanted the piece to remain in a fairly conventional tonal space. The most effective way I found to do this was to control the two chord synths with a pseudo-arpeggiator, which automatically generated a familiar chord structure given a note and a quality.

When dealing with the data sets, I noticed that some points of the data were not as precise as I would have liked. For example, the swell height measurements in meters were only measured to one significant figure, so there were often long periods of constant data, which didn't make for interesting modulation. To remedy this, I created a system which employed a variety of data manipulation methods in order to make it grow and change more gradually. I employed similar methods when writing the notes to the score, smoothly transitioning between each "bar," or each data point.

Every time the program is run, the latest data set is downloaded and the piece is composed anew using this data. In theory, it would be different every time - although there wouldn't really be any change if you run it two times within a few minutes of each other, if you were to play it the next day, there would be a distinct difference. It takes the listener through about 45 days of the weather in the Cape Cod Bay, and if the seas were rough at any point, it's definitely evident in the music.

Technical Description

The program starts by trying to download the data, using the class WaveDataFetcher to do so. This class reads a text file published on NOAA's website that contains the raw data collected from the buoy, which is updated every 30 minutes. The data must then be formatted from raw lines of text into a more usable format, so the first stage of formatting is done by parsing each data point into its components and storing them in an ArrayList of WaveDataPoint instances. Each WaveDataPoint instance has public member variables that represent all the types of measurements that the buoy collects, from a timestamp to the overall

wind direction. Some error correction is also done at this stage - if there is missing data at any point, the WaveDataFetcher repeats the last valid data point until a new valid data point is entered.

Once the data is in a usable format, it can now be parsed into a format suitable for driving the composition. This is done in the WaveDataParser class. The primary goal of this class is to assign each metric to some parameter of the synth, or to use multiple metrics to control a single parameter. All of this parameter information is stored in DataLists, which extend the ArrayList class. There are four DataLists in the class which represent the three instruments and some miscellaneous modulation, and they each contain a different type of node. There are three node types: ChordNode, BassNode, and ModNode. Each of these nodes contain the parameter information for their respective synth at that data point.

The ChordNode type contains values for amplitude, arpeggiator period, pitch and chord quality. For starters, the amplitude is set by the wave height, and the arpeggio period is set by the overall wave period. On the buoy, there are two types of waves measured - swells and wind waves - so one voice is driven by the swell data, and the other is driven by the wind waves. For pitch and quality, I needed a more robust system. I wanted the wave direction to control the pitch such that when the wind waves and swells were opposing each other (implying rough weather) then there would be more dissonance. The most obvious way to convert direction (a circular metric) to pitch is to map the directions directly onto the circle of fifths. This would achieve some nice consonance when the directions of the two wave types were similar, and would be very dissonant if they were different. The issue with this is that the measurements for wave direction are only given in compass rose directions, of which there are only 16 (“NNE” or “WNW” is as precise as it gets). I tried at first to throw out four measurements and interpolate new values in between, but it proved to be not very true to the data and not particularly musical either.

The solution I came up with was to find the lowest common factor between the 16 compass rose points and the 12 notes in the circle of fifths, which is 48. At 48 discrete points, I can use sub-indices within the original 12 notes to denote the quality of the chord - if it falls on the left side of the note, it is minor, leaning towards a subdominant-dominant motion, and if it falls on the right side of the note, it is dominant, leaning towards a dominant-tonic motion. The other two sub-indices within the note make a major chord. So, for example, a value on the circle of 9 would produce a D minor chord (3rd note, 1st sub-index). This works well in theory, but with only 16 possible input values from the compass rose, there would never be any chord qualities other than a major chord. To remedy this, the wave direction is averaged with the wind direction, which is measured in integer degrees, giving a wider range of possible values, which are then re-quantized to the 48 value circle of fifths.

The BassNode type contains values for pitch and filter frequency. The pitch of this synth is simply the lower root of the two chord synths, pitched down several octaves. The frequency of the synth voice’s internal lowpass filter is controlled by the “steepness” measurement from the buoys. There are only a couple possible values for steepness, from “AVERAGE” to “VERY_STEEP,” so arbitrary values between the minimum and maximum cutoff of the filter are chosen for each value. The steeper the waves, the more open the filter, and thus the more gritty the bass sounds.

The ModNode type contains values for reverb amount, modulation period, and random seed. The reverb amount isn’t really a reverb amount - instead it controls the percentage of feedback on the delay circuit. It is controlled by the average wave height metric. The modulation period affects the frequency of the LFOs controlling the frequency of each chord voice’s lowpass filter, and is controlled by the average wave period metric - so the filters sweep up and down in a realistic wave-like motion. The random seed is simply the main wind direction, which is used as the parameter for a call to JMSLRandom.setSeed() in each new measure. Random calls are used to get random

values from the arpeggiator, among a few other things, so in this way the wind direction influences the actual makeup of the notes on the staff.

All four DataLists are populated with their respective node types using these mappings. After the lists are populated, the data must be further processed in order to be consistent and usable. The first step is to smooth the data using interpolation and weighted moving average smoothing. The smoothing process must be run for each parameter that needs it. For each parameter, there are two passes through the data for interpolation and one pass for weighted moving average. The first interpolation pass searches for repeats in the data and creates a truncated list of index/value pairs that removes long repeated sections of data and replaces them with a single value, with an index equal to the midpoint of the repeated section. The truncated list should not have any repeating sections. The second interpolation pass then fills in the resulting gaps in the data using cubic interpolation. The third pass through the data calculates the weighted moving average and applies it to the data. I found that simply replacing the existing data with the weighted moving average of the data changed it too much, so the final values after this pass are actually linearly interpolated from the output of the interpolation passes, using a mu value that can be different for each different parameter.

After smoothing, the range of values may still not be usable to drive a JSyn input, since JSyn inputs tend to have very specific usable values. Therefore, many data points are then normalized to fit within usable ranges. Some values, such as the filter frequency for the bass voice, are mapped to a certain range, and others are mapped to just positive numbers, like the arpeggio period of each chord voice. This normalization allows the data to conform to any range deemed necessary for the composition.

Now that that data is completely formatted, the actual composition is created. The composition itself is done within JScore, using JSynUnitVoiceInstruments for each voice. The delay is also included as a JSynUnitVoiceInstrument, so for each measure a note has to be drawn in order to get any sound out of the delay line. I opted to not use Orchestra Patch objects to patch everything into the delay, so that each voice would also have a dry output in the mixer. Before generating a new measure, the random seed of JMSLRandom is reset using the random seed parameter. The tempo of each measure is 15 BPM, which is 4 seconds per beat. The function getArpeggioPitchSet is used to get the actual set of MIDI pitches that will be randomly selected to form an arpeggio pattern for the chord voices.

When there is no smoothing applied within each measure, there can be a noticeable jump between parameters, mostly the arpeggio period, when transitioning between measures. To prevent this abruptness, the duration of each arpeggio note is linearly interpolated between the current measure's parameter and the next measure's parameter. A couple of preventative measures need to be taken to avoid asymptotic behavior within the interpolation function, including setting a relatively high minimum possible note duration. Other than this extra smoothing, the rest of the notes are generated fairly normally using the parameters in each node of each DataList.

Once the notes are all generated, the piece is essentially "done". The score window will open and the piece can be played and scrolled through, so the listener can have a preview of what will come. The low bass notes don't really display properly, but they sound perfectly fine. It would probably be better if the notes in the chord synths were beamed as well, but JMSL has an issue with the beaming of notes on occasion, so it's fine for the time being.

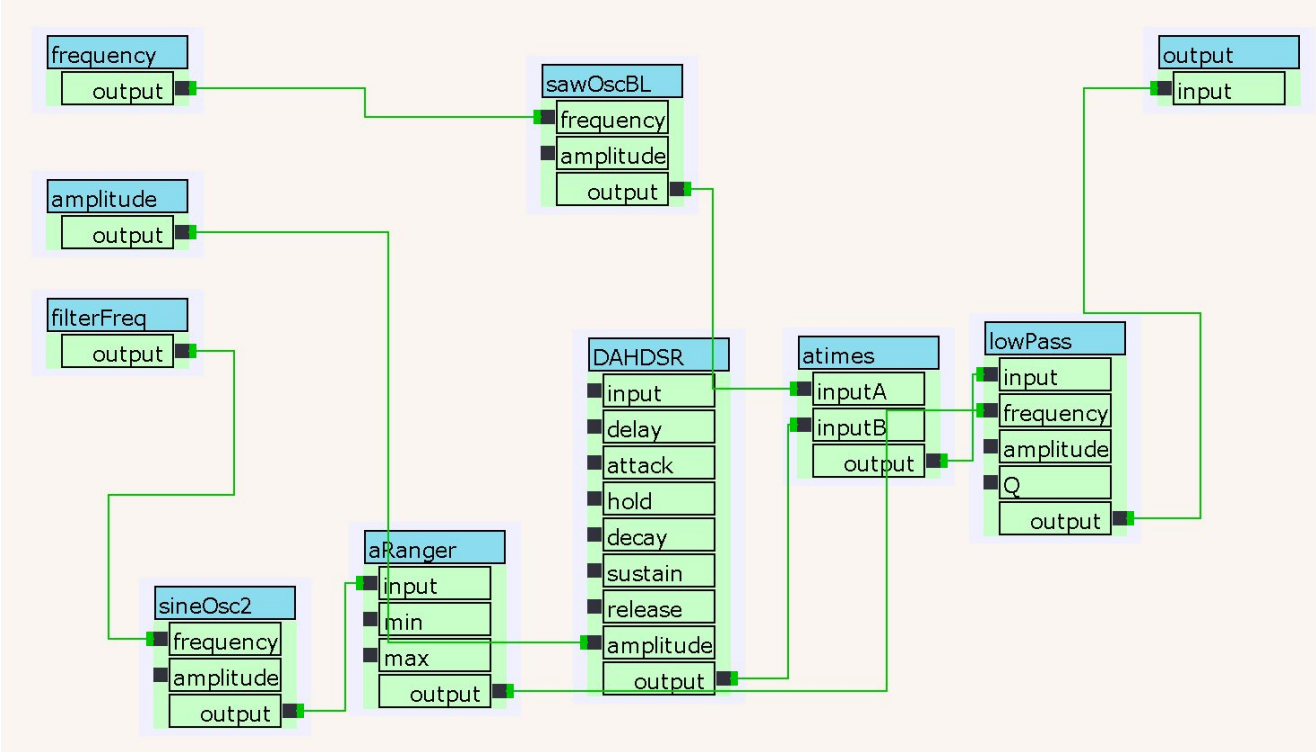
Artistic Analysis

I'm unsure whether or not my piece was actually a valid representation of the weather patterns in the bay. Given my implementation of the data, it might not actually be dissonant when the weather is rough, and calm when the weather is calm. I think there just wasn't enough metadata involved to inform the process. However, I think I did succeed in creating an interesting piece of music - by sticking primarily to

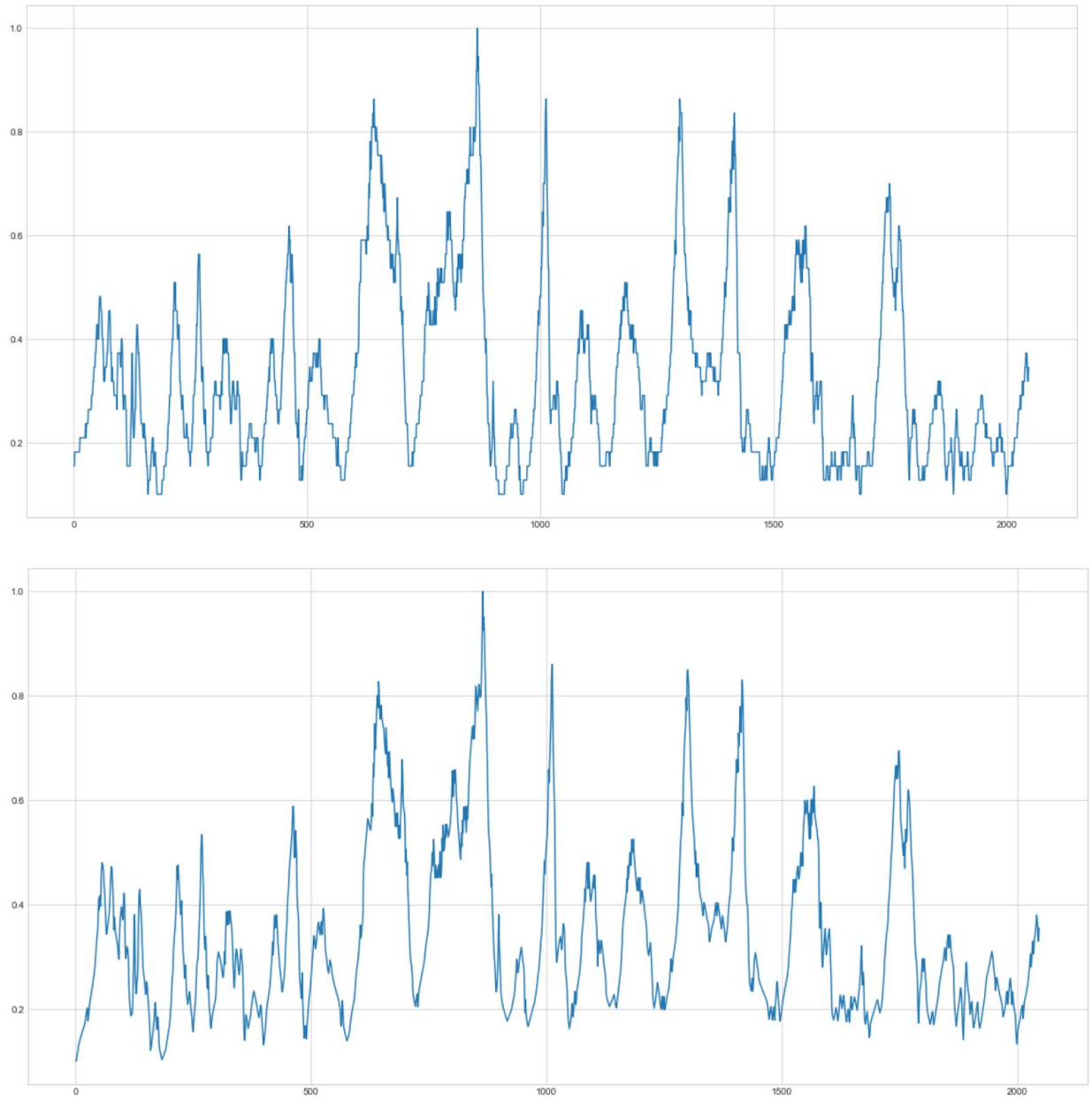
tonal structures that were predefined, and letting the data control them, I was able to create a fairly pleasant piece that didn't need a bunch of manipulation to make work. I was actually surprised by how easy it was to make it sound good, and even interesting. I'm definitely interested in improving the parameterization of the data in the future, and creating a real-time constant generator that uses current data to make music constantly, instead of writing a score. I also know that my interpolation and normalization methods could be significantly optimized and improved, which could in turn allow for the data to be more useful in driving the composition. I think that I actually made it too long as well - two hours doesn't really give the average, easily distracted user time to experience the massive range of timbres and sound qualities that the piece has to offer.

Conclusion

The ability that JSyn and JMSL have to utilize any possible input to change a sound was made very clear to me in this project. It was eye opening to see how easy it is to drive sounds with just about anything. I also quite enjoyed the data processing portion of the project; I believe sonification of data has become a new passion of mine. I think it would be interesting to implement this project in a real-time context which can be accessed on the web, so that any listener can tune in and listen to how the seas are doing in the middle of the Cape Cod Bay, from anywhere. I think that would be a nice thing to have, even for people who aren't as close to it as I am.



Syntona patch for one of the two chord voices. The other one uses a pulseOsc instead.



Wind wave height data before and after processing.Note the elimination of repeated sections and the zero offset added by the weighted moving average function. The peaks are still intact but are less quantized.

Sources

NDBC Glossary of Terms:
https://www.ndbc.noaa.gov/station_page.php?station=44090&uom=E&tz=STN
Raw data from the buoy:
https://www.ndbc.noaa.gov/data/realtime2/44090.spec
Interpolation methods:
http://paulbourke.net/miscellaneous/interpolation/